What is a firmware library?

A firmware library is a ready-made helper code that knows how to talk to hardware for us.

Instead of us controlling hardware *bit by bit*, the library gives us **easy function,** we need not understand the core electronics to program it.

We want a light of the LED to be lit up, we just call the function in the library, instead of telling the microcontroller, to which pin I should give the electricity to and how much inorder to light up the LED on the board.

From what I understand, is that everything on the board is either ON or OFF its just electrical signals timed properly to get the proper output. The microcontroller turns tiny electrical paths on and off in a very precise order and at very high speed, this is the very basis of all the computation and control. Firmware and Firmware Libraries decide when and how long each signal should be ON or OFF ensuring the hardware behaves as intended, and we as programmers can focus on building the actual Application Logic or the tasks to be performed.

Lets take Logic Gates for example,
We know the basic ones such as AND, OR, NOT etc, and if I have to build a bigger circuit like a ADDER or MULTIPLIER I would think of the basic gates. If I know I can build some algorithm using the MULTIPLIER I would use the already built one as I know the logic works, I wouldn't start building it from scratch, similarly if I have to build a working prototype of my robot I would prefer make a Arm work in different way in different situations by using sensors and controlling using microcontrollers instead of designing my own sensor firmware…

Other things I learnt:

FirmwareFlow

All about Firmware

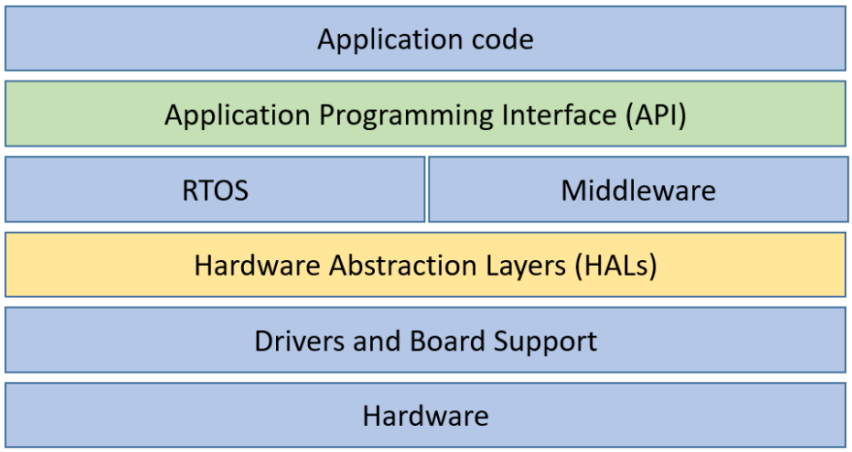
FlashMemory

What is inside a Microcontroller ?

RISC-V

Why APIs are important in embedded systems?

At the lowest level, embedded hardware is controlled through device registers, bit masks, timing constraints, and electrical rules defined by the datasheet.
If I don't use APIs or firmware libraries, I would have to think of all the edge cases, take care of the timings, voltages, current, everything. That's not efficient, since the world works in a modular approach i.e, we just have to focus on getting the task done without worrying too much about how the internals of it works, then the work gets done quickly and we'll have a clarity of thought.
If we try to dig deep, there are high chances we will lose the bigger picture and it would consume a lot of time and manpower to do the same task.



Firmware is the complete package of all of the above layers together and flashed to a device's Flash memory.

What was understood from the lab code?
In short what ever codes we write we write it in a modular approach so that it is readable and there can be further work done on that.

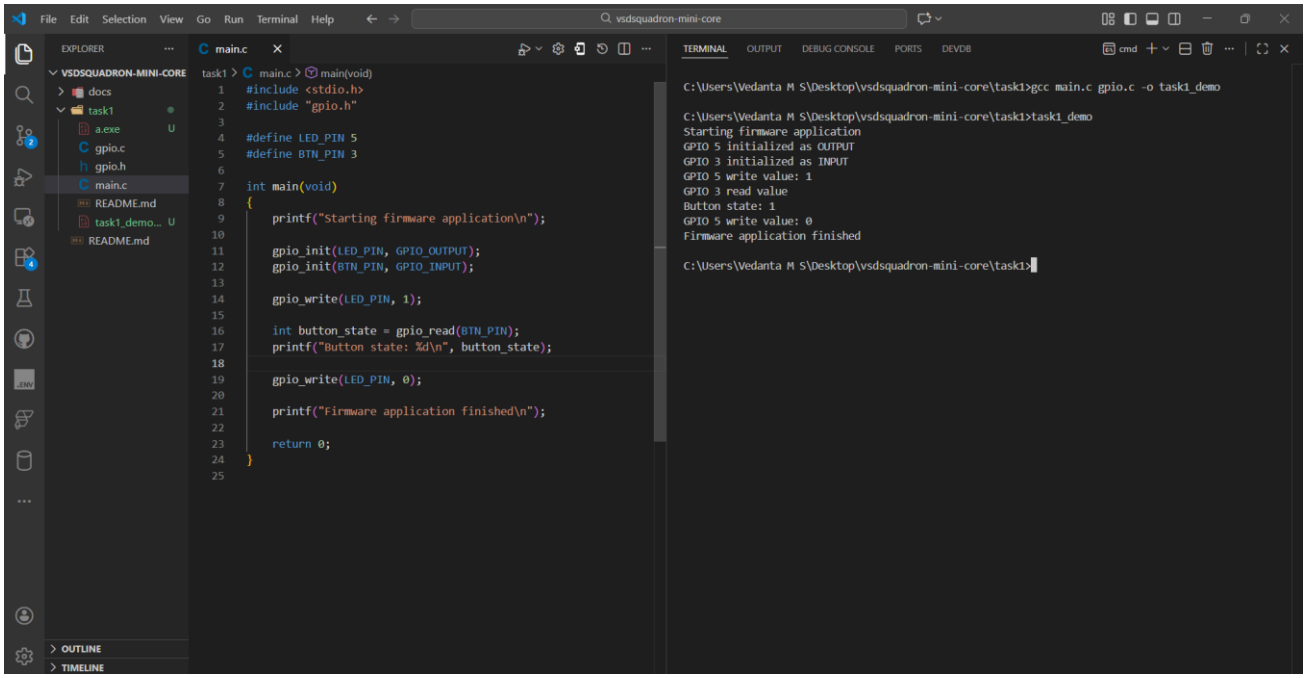
gpio.h - There are header files which defines the basic structure of the functions, which is called the INTERFACE. There is no logic written, it is just the name of the functions and how they should be called.

gpio.c - There is another file which describes the implementation of the functions present in the .h files

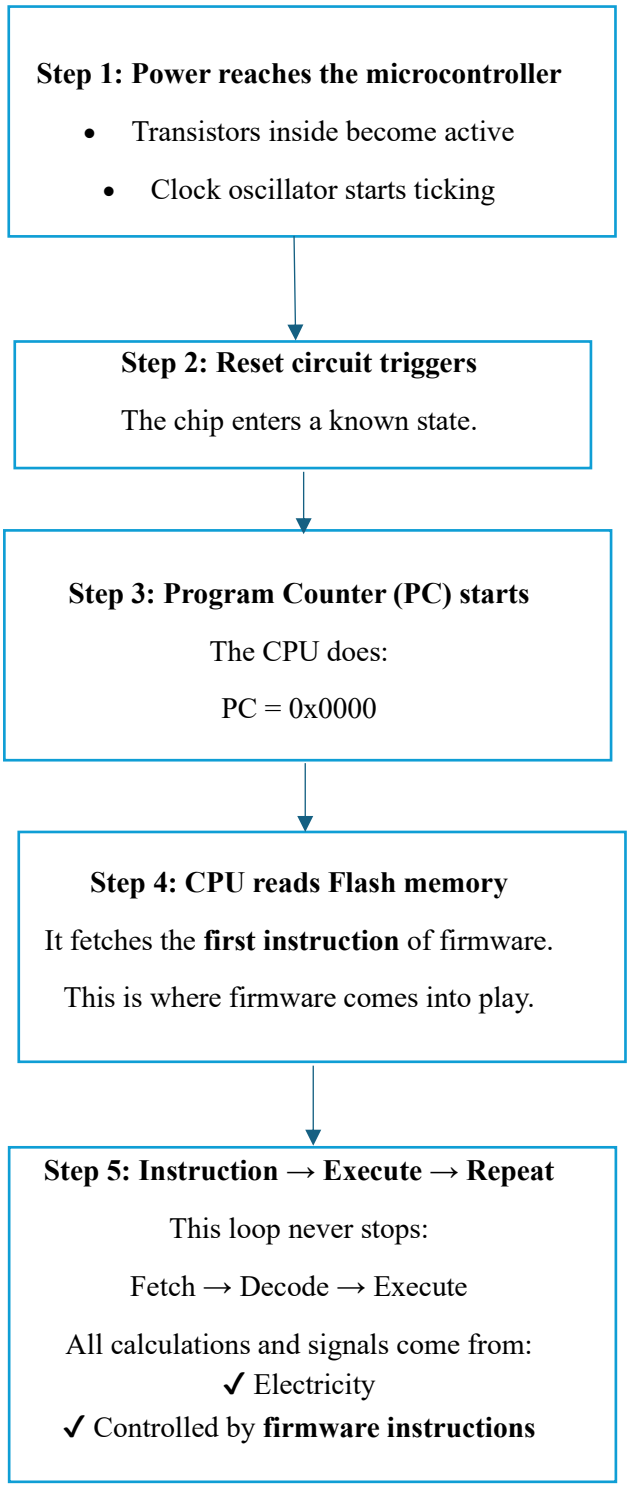
Main.c – this has many printf statemets which I understood from gpt that it was to simulate hardware behaviour, i.e instead of doing a PRINTF to the terminal we would actually write it to the registers.

One more important concept was – HAL (Hardware Abstraction Layer) that is implemented here, gpio_write or writing to a particular register could be done directly on the hardware rt ?
Yes it can be but what if you change the Microcontroller or change the Pin mapping ? we would have to rewrite the whole code. Instead we just keep the codes modular so that the code is readable,

portable and testable.
If a new engineer wants to correct it he can do it easily.



What Happens when you Power VSD Squadron Mini Board ?



Firmware

What is *firmware*?

Firmware is the **software that runs directly on hardware**.

- Runs on microcontrollers (Arduino, ESP32, STM32, etc.)
- Controls **pins, sensors, motors, communication**
- Lives *between hardware and your application logic*

What do you mean by “BETWEEEN”? Is it physically between ? or is the logically between or between the code lines ?

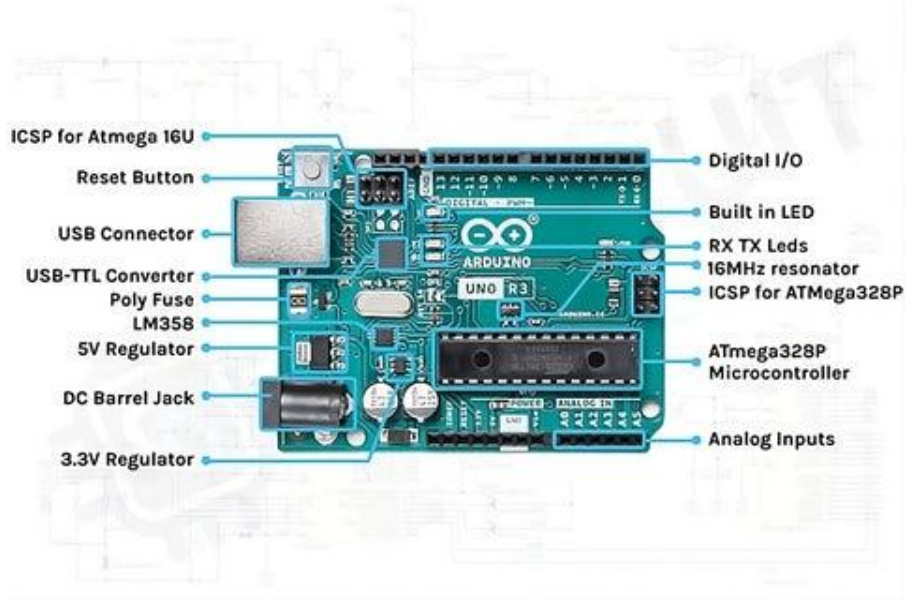
At the end of the day whatever wherever the code is… it should help us control the hardware.
Also we know that, Software – is set of instructions

Therefore, Firmware is a set of instructions that directly runs on hardware.

Hardware ? you mean registers, capacitors, transistors ?

No Firmware is run by the Microcontroller’s CPU unit which in-turn produces signals which uses other parts of the PCB such as registers, capacitors, resisters etc…

Control the hardware? How can you control the hardware?
Lets take a simple example of Arudino UNO and understand this



Let’s start our story from the time the Arduino Board gets the electricity . Because that is where the Hardware gets its LIFE isn’t it ??

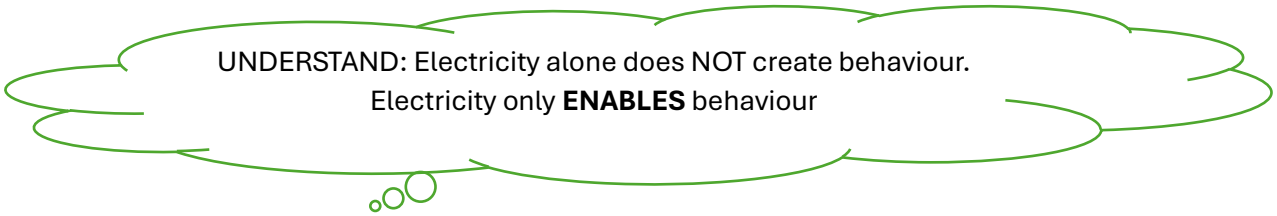
Step 1: Powering the board

What happens when you power an Arduino board?

YES — electricity flows to:

- Voltage regulator , Capacitors, Resistors, LEDs
- Microcontroller (ATmega328P on Arduino Uno or CH32 in case of VSD)

All components that are connected to the board **do get powered**.



(Its not like a light bulb, if I turn the switch ON the light bulb brightens. Yes the thinking is correct but light BULB is a Passive hardware - No decision-making, No logic, No memory, No timing)

Behaviour is fixed by physics

Power ON → Light ON

Power OFF → Light OFF

That's it.

Arduino is NOT like a bulb

Arduino contains a **microcontroller**.

A microcontroller is:

- A tiny computer
- With memory
- With logic
- With a clock
- With decision-making ability

Thing	**Role**
Electricity	Enables operation
Microcontroller	Executes logic
Firmware	Defines behavior

But here's the **critical truth**:

A microcontroller does NOTHING useful unless it is told **what to do.**

Application Code – "Tells WHAT to do?"
Firmware – "Tells HOW to do it using the hardware we currently have?"

Step 2: Microcontroller wakes up

The ATmega328P:

- Receives 5V
- Clock starts ticking (16 MHz)
- CPU comes out of reset

OK now the CPU is alive as it got its electricity. It's only PURPOSE OF LIFE is to serve the almighty so when its alive it asks for INSTRUCTIONS – "What should I do ? What instructions should I execute? "

SO now Who is that Almighty which gives the CPU or the Microcontroller its Instruction ?

The thing is, the Instructions is already written in a memory called FLASH memory which is present inside the Microcontroller.

Where does it get instructions from?

From firmware stored inside it

Inside the microcontroller is:

- **Flash memory** → stores firmware
- **SRAM** → variables
- **EEPROM** → saved data

The firmware is NOT in resistors or capacitors
he firmware is stored **inside the microcontroller chip itself**

Then why doesn't the board "do something" automatically?

Because **hardware by itself is dumb**.

Example:

- A resistor only resists current
- A capacitor only stores charge
- A transistor only switches
- A microcontroller is just **silicon gates**

⚡ Without instructions:

- No blinking

- No calculations
- No decisions

What makes the Arduino “think”?

Firmware

What exactly is firmware (physically)?

Firmware is:

- **Binary machine code**
- Stored in **non-volatile memory (Flash)**
- Inside the **microcontroller**

Where does Arduino firmware come from?

You write:

digitalWrite(13, HIGH);

This becomes:

- C++ code
- Compiled to machine code
- Uploaded via USB

Stored in:

📦 **Flash memory inside ATmega328P**

What is the bootloader?

The bootloader is a **tiny firmware**, pre-installed at the factory and lives in a protected Flash area

Its job:

- Listen on USB/Serial
- Accept new firmware
- Write it into Flash

After upload:
➡ Bootloader jumps to **your firmware**

Under the hood:

digitalWrite()

└── Arduino core firmware library

└── Sets specific MCU register bits

└── Electrical signal appears on pin

Firmware libraries:

- Translate **human-friendly functions**
- Into **hardware register operations**

Why don't we write registers directly?

You *can*:

PORTB |= (1 << 5);

But firmware libraries:

- Prevent mistakes
- Handle timing
- Work across boards
- Save time

Electricity: Enables the hardware

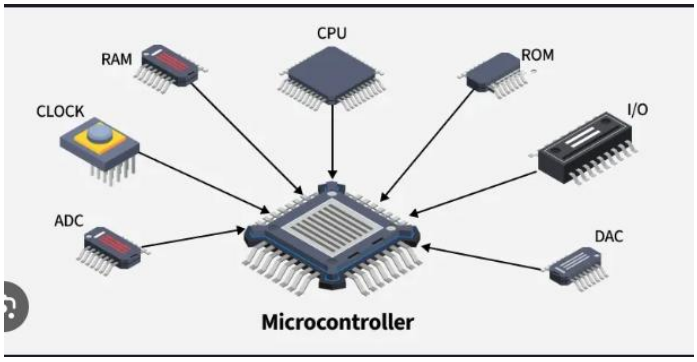
Microcontroller: Executes instructions

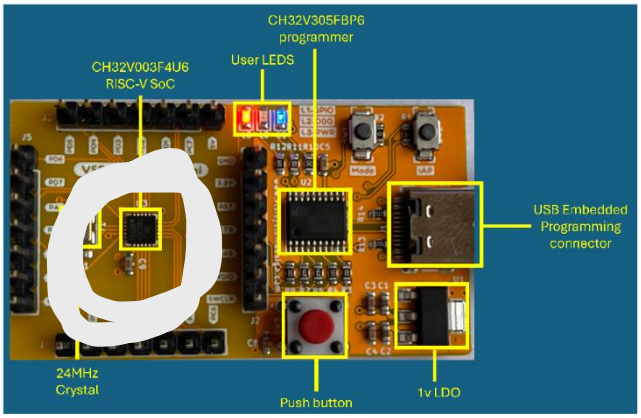
Firmware: Instructions stored in Flash

Firmware libraries: Reusable code that controls hardware safely

What is a microcontroller?

A microcontroller unit (MCU) is essentially a small computer on a single chip. It is not just a computing unit it is the whole computer. But it doesn't need an OS to operate.





1. **Central Processing Unit (CPU)**: Acts as the brain of the microcontroller, executing instructions and performing calculations.

2. **Memory**: Stores data and instructions for the CPU to access.
   i) **Flash Memory**: Non-volatile memory used for storing the program code.
   This is where the Firmware is Present.

   ii) **SRAM (Static Random Access Memory)**: Volatile memory used for temporary data storage.

   Example: A microcontroller might have 256KB of flash memory and 64KB of SRAM.

3. **Input/Output (I/O) Ports**: Allow the microcontroller to communicate with external devices.

4. **Peripherals**: Additional hardware components that extend the microcontroller's functionality.
5. **Power Management**: Components that manage the power supply to the microcontroller

6. **Clock**: Provides a timing signal that synchronizes the operation of the CPU and other components.

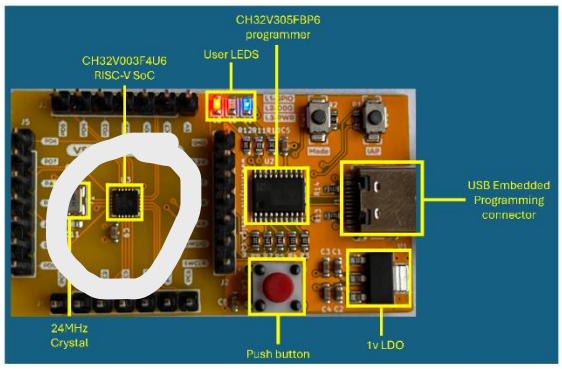
Flash Memory

What is flash memory *physically*?

Flash memory is made of **floating-gate transistors**.

Each bit:

- Stores **charge trapped inside silicon**
- Keeps data **even when power is OFF**



Inside the microcontroller, flash is just a **big linear array**.

Flash is a Non-Volatile Memory and is present inside the microprocessor

How flash is divided internally ?

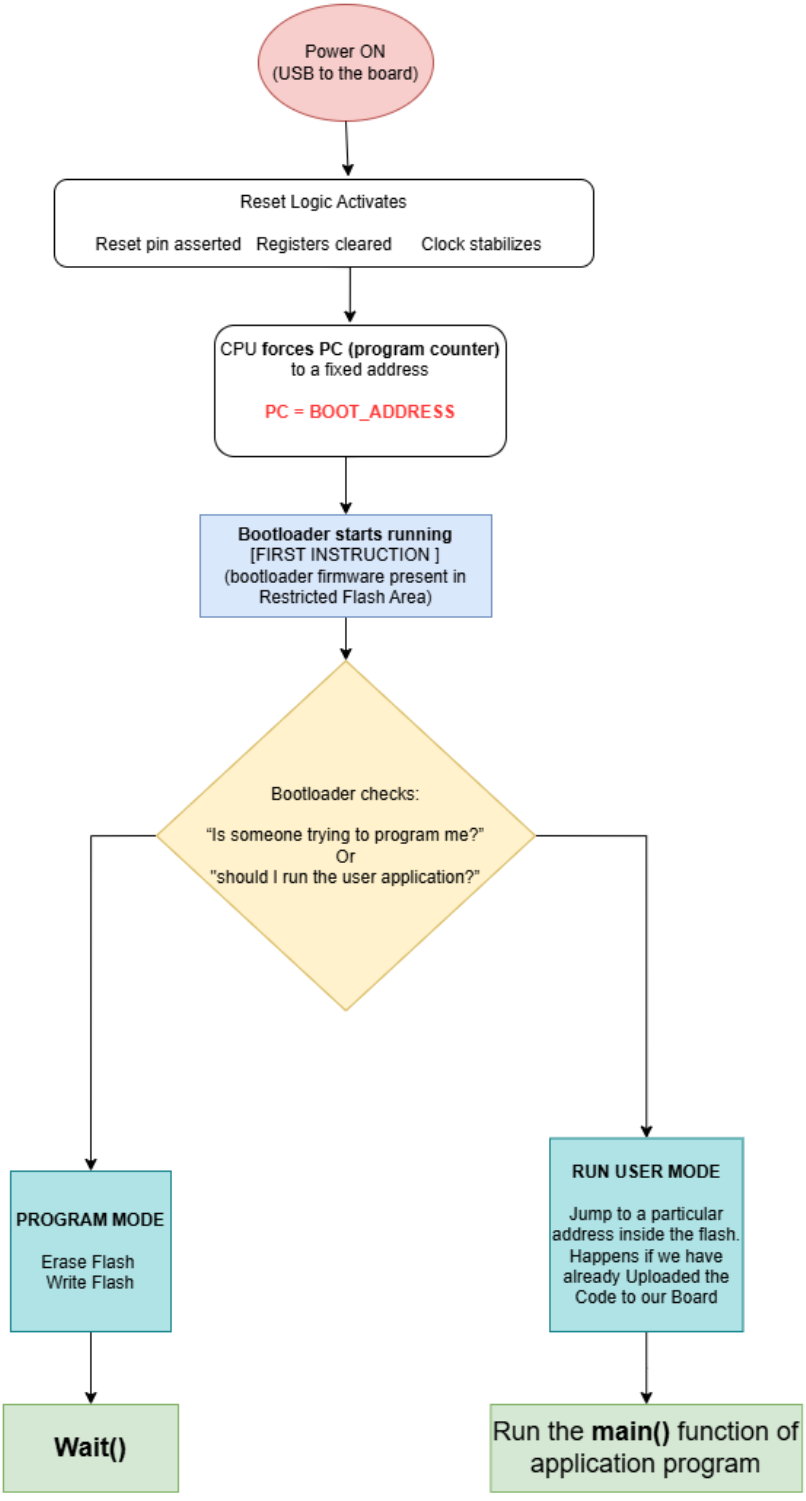
Typical layout on the VSD Squadron Mini looks like this (simplified):

vbnet Copy code

```
┌──────────────────────────────┐
| Bootloader (RESTRICTED)      |  ← protected region
| ~1.9 KB                      |
├──────────────────────────────┤
| User Application Flash       |  ← your code
| ~14 KB                       |
├──────────────────────────────┤
| (Optional config / option bytes)
└──────────────────────────────┘
```

The restricted part of flash memory permanently stores the Bootloader, which is the very first code the CPU executes after reset, and hardware protection prevents your application from overwriting it.

What happens when power is applied?



RISC-V

RISC-V is an instruction set architecture (ISA).

It basically defines what instructions the CPU understands. Think of it like a language. We humans understand English, CPU understand RISC-V instructions.

What is ISA?

A set of instructions the CPU understands

Instruction	**Meaning**
ADD	Add two registers
SUB	Subtract
LOAD	Read memory
STORE	Write memory
JUMP	Change execution flow

If an instruction is **not in the set**, the CPU **cannot execute it**.

RISC-V is a specification, not a chip and not a language.

It says -

“If you build a CPU that claims to be RISC-V, it must understand these instructions and behave exactly this way.”

What about RISC-V Assembly?

RISC-V Assembly is:

- A **human-readable representation** of RISC-V instructions

- One-to-one mapping with machine code

Example:

ADD x1, x2, x3

Internally becomes

0000000 00011 00010 000 00001 0110011

The CPU only sees **binary**, not text.

So firmware:

Is **written in C / C++** → Compiled into **RISC-V** machine instructions → Executed by a **RISC-V** CPU

Why can't we call RISC-V a programming language?

- No variables
- No data types
- No loops (only jumps)
- No functions (only calling conventions)
- No memory safety
- No abstractions

It is a hardware control language, not a human problem-solving language

In Short: **RISC-V is not a programming language; it is an instruction set architecture that defines the exact machine instructions a CPU understands, and all firmware must be compiled into those instructions to run on a RISC-V processor.**